A Fault-Tolerant, Scalable, Low-Overhead Distributed Garbage Detection Protocol

Marc Shapiro

A Fault-Tolerant, Scalable, Low-Overhead Distributed Garbage Detection Protocol

Marc Shapiro
INRIA, BP 105, 78153 Rocquencourt Cédex, France
shapiro@sor.inria.fr

June 1991

Abstract

We present a protocol for the distributed detection of garbage in a distributed system subject to common failures such as lost and duplicated messages, network partition, dismounted disks, and process, site and disk crashes. The protocol uses only information local to each site, or exchanged between pairs of sites; no global mechanism is necessary. Overhead is low. The protocol is parallel and should scale to extremely large systems.

1 Introduction

Recent development of the object-oriented technology has sparked interest in low-level support systems for user-defined objects. A number of operating systems [2, 6, 11] and database systems [5, 8, 13] offer such support. One important aspect of objects is that one may contain *references* to other objects. A program's activity creates objects and modifies the references between them; an object for which no reference remains has become inaccessible *garbage* and could be de-allocated. Automatic garbage collection (GC) is a valuable service, as it frees programmer resources and is safer than manual collection.

Many published distributed GC algorithms are based on very strong assumptions and/or expensive, non-scalable mechanisms, making them unsuitable for a low-level object-support system. In contrast, we propose a protocol for distributed garbage detection based on reasonable, weak assumptions. Messages may be lost, delivered out of order, or duplicated. Nodes may crash. Objects may migrate or be deleted.

The protocol is fully parallel, and bases itself only on local and pair-of-sites information. Since no global mechanism is necessary, it should scale to any number of nodes. No assumption is made w.r.t. the semantics of objects; for instance any object may either persist or disappear after a crash.

Our limiting assumptions are that crashes are fail-stop, and that messages are either lost or delivered unmodified in finite time. We consider both reliable and unreliable communication media. We only consider passive objects.

Here is the basic idea of our protocol. Each disjoint *space* maintains a list of potential incoming and outgoing references, called respectively the Object Directory Table (ODT) and External Reference Table (ERT). Both the ODT and the ERT are conservative estimates. Local garbage collection proceeds from the union of the local root and the ODT and remove entries in the ERT, which in turn allows previously-pointed-to ODTs to be collected. A separate subprotocol deals with inter-space cycles of garbage.

Since local GC starts from the union of the local root with the (conservatively estimated) ODT, all non-reachable local objects are true garbage. Each local GC cleans the ERT of useless stubs. In turn, ERTs are used to clean the ODTs, yielding successively better estimates.

The paper proceeds as follows. First, Section 2 discusses the principles of garbage detection and collection, and previous work on distributed GC. Then Section 3 describes our model and notations. Section 4 presents the protocol and its subprotocols. In Section 5 we informally argue that the protocol is correct. In Section 6 we give a qualitative characterization of its performance. Finally, Section 7 concludes.

2 Principles of distributed garbage detection, and previous work

The purpose of garbage detection is to distinguish objects accessible from a so-called *root*, from others which are called *garbage*. Classically one distinguishes

two rôles: the *mutator* and the *collector* [3]. Dividing the work of the collector into two parts, garbage detection and garbage disposal, we will speak of the *detector* rôle.

There are two well-known families of garbage collection algorithms. Reference counting algorithms attach a counter to each object, maintaining the (very strong) invariant that the value of the counter is at all time equal to the number of references to the object. This is inherently hard in a distributed environment, especially with failures. We will not consider reference counting algorithms any further. Note however that in the case where messages are reliable, an ODT entry (defined later) degenerates to a local reference count.

In tracing, the detector performs a walk of the "refers to" graph, starting from the root; at the end of the graph traversal, any objects not reached are garbage.

In Vestal's [14] tracing algorithm, the universe is divided in "areas", in which parallel collection may occur. It is characterized by a high space overhead, and does not take advantage of locality: each collector performs a global transitive closure starting at the root of one area.

Hughes [7] proposes a garbage collector for a distributed-memory multiprocessor with a single clock. Each area has its own local root; a detector repeatedly starts at each local root. A detector which starts from some root at time t marks all objects it reaches with the value t. Later markings supersede earlier ones. Thus, the date on reachable objects will be constantly be advanced; however once an object becomes unreachable, its date mark will not change anymore. Objects marked with a date less than some global minimum are collected. The simplicity of Hughes' algorithm is quite appealing. Note however that determining the global minimum requires a global termination algorithm. Furthermore, if a single processor is disconnected (i.e. communication is impossible with it), it is impossible to advance the global minimum, which completely disables distributed collection.

Liskov and Ladin [10], describe a fault tolerant distributed garbage detector based on a highly available service, which is logically centralized but physically replicated. Nodes may crash (fail-stop) and recover, messages may be lost or delivered out of order. All objects and tables are assumed backed up in stable storage. Clocks are synchronized, and message delivery delay is bounded. The distributed garbage detector relies on local tracing garbage collectors informing the centralized service about its references to remote objects. Local collectors query the centralized service about the real accessibility of their public objects to better estimate their root.

Dead inter-site cycles are detected by the centralized service. Based on the paths transmitted, the centralized service builds the graph of inter-site references, and detects dead cycles with a standard GC algorithm.

In her thesis [9], Ladin simplifies, and corrects the deficiencies of, the above proposal by adapting Hughes' algorithm (with some useful optimizations). Instead of a single clock, she assumes synchronized local clocks. The centralized service determines the global minimum date, making a termination protocol unnecessary. It is no longer necessary for the centralized service to detect cycles, since Hughes' algorithm takes care of them.

3 System Model

This section gives some definitions and our notation, and lists our assumptions. These are quite reasonable and minimal, and do not restrict the generality of our protocol.

3.1 Objects

Objects are passive entities. An object may contain any number of references to other objects. We are not concerned with the semantics of a reference; we simply assume that a reference names its target in a way meaningful to the application.

References to deleted objects are allowed. A deleted object contains no data and no references.

3.2 Spaces

The objects is partitioned into disjoint *spaces*. At any time, an exisiting object is either located in some space or in transit (migrating) between spaces.

References within a space are assumed much more common than across spaces (locality principle). We use the word space (rather than host or machine for instance) to abstract away from the different kinds of subdivisions found in distributed systems. In some systems a space is a process or a storage volume, in others a computer, in others a local network. In one of our implementations, a space is a logical object container spanning multiple processes and computers.

Each space has its own local root. Each space performs a standard local tracing garbage detection, independently of the activity of remote or global detectors.

The root for local collection is the union of the local root and the ODT (see below). Any correct tracing GC algorithm is suitable locally.

The global root is conceptually formed of the union of all local roots, but this global root is in fact never used, only local information being necessary.

We distinguish between a *local* reference (to an object known to be in the same space) and a *remote* one (to an object thought to be in another space). A remote reference is represented as a local reference to a *stub* object, which contains the last known location of the referent. Stubs are collected in a structure called the *External Reference Table* or ERT. The ERT is maintained by the local collector and readable by the global detector.

Conversely, local objects potentially accessible from other spaces are listed in a *Object Directory Table* (ODT). The ODT of a space is maintained by a loose cooperation between the global detector (from information contained in remote ERTs) and the local mutator. A mutator is allowed only to add ODT entries, not to to read or remove them.

A space may terminate. Normal termination causes all its contained objects to be deleted normally, and all indirections crossing it to be eliminated. A space may also terminate abnormally; in this case, a protocol described in Section 4.8.2 recovers to normal termination. Crashes are asssumed fail-stop, i.e. a crashed space ceases to compute and to send messages, but otherwise takes no action.

3.3 Communication

Communication between mutators in different spaces occurs via messages. A message may contain both references and objects; the objects it contains migrate from source to destination space. Our protocol adds a few *control* messages to the normal *mutator* messages.

Two spaces A and B are connected by a single channel. Our garbage detection protocol tolerates message loss and duplication. In this paper we consider FIFO channels only; out-of-order delivery is considered in [12].

3.4 Finder

To use an object reference it is necessary to track the corresponding object's location. This function is performed by a system component called the "finder" by Fowler [4]. The finder's operation includes "path compression", i.e. the elimination of chains of indirect references, via multiple ERT/ODTs. This is beneficial to both the mutator's and the garbage detector's performance. Conversely, the finder benefits from the collection of garbage location information.

For the sake of brevity, we will not elaborate here on the interactions between the detector and the finder; these are detailed in [12].

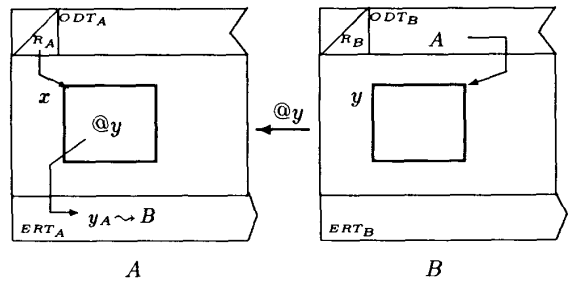

Figure 1: Notation

3.5 Notations

The following notations (summed up in Figure 1) will be useful for the rest of the paper. We note spaces with capital letters A, B, C, etc.; objects with lowercase letters x, y, z; and references to such objects as $@x$, $@y$, $@z$. The ODT and ERT of space A are noted ODT_A and ERT_A. The local root of A is R_A. The stub object, on space A, for remote object y, thought to be in space B, is noted y_A, and contains the location information $\rightsquigarrow B$. A $@y$ on space A will in fact point to y_A.

On space B, an ODT_B entry, at some index i, will contain the information that y is potentially referred from space A, which we note:

$$ODT_B[i] = (A@y)$$

In Figure 1, object y is not reachable from B's local root R_B, but in fact is live, being reachable (via x) from R_A.

Figure 1 also shows a mutator message from B to A containing $@y$, noted

$$B \rightarrow A : \mathsf{mutator}\{\ldots, @y, \ldots\}$$

In the case of a control message, the word mutator is replaced by the specific control-message type.

4 The Protocol

We will now describe the protocol and its application. The detection protocol is based on local marking

garbage collectors, and on interaction of a number of sub-protocols: the finder protocol (omitted here), a reference-sending protocol, an object-migration protocol, an interaction protocol between local collectors, and a cycle detection protocol. Furthermore, fault-tolerance refines the finder with a deletion protocol and a termination protocol.

Although the basic idea (given in the Introduction) is simple, the actual details can be quite complex. Here we propose a version for distributed systems with failures (lost and duplicated messages, disconnection, and crashes). To simplify the presentation, we make the (unrealistic) assumptions that message delivery is instantaneous, and that messages are delivered in the order they were sent. We refer the interested reader to reference [12], which specifies the full protocol, considering non-instantaneous and out-of-order delivery.

4.1 Initial State

Let us start the description for some instant where ODT_A and ODT_B contain an exact (i.e. minimal) description of their incoming references. One such instant is when a space is initially created, and its ODT is empty.

Let us run A's local GC; it will trace all local live accessible objects from the local root R_A and exact ODT_A. Garbage stubs will be deleted. At the end of the local GC, ERT_A contains an exact image of the outgoing references.

4.2 Sending a Reference

Whenever A's mutator sends a message

$$A \rightarrow B : \mathsf{mutator}\{\ldots, @z, \ldots\}$$

containing a reference $@z$ to B (see Figure 2), we consider that a *potential reference* is created from B to A. *Before* the message is transmitted, an ODT entry

$$ODT_A[j] = (B@z)$$

is created (or located if it already exists). If two different spaces possibly refer to a single object of A, each will be assigned its own ODT entry.

Note that a single ODT entry is created, whether or not z is local to A, whatever number of references are thought to point from B to A, and without knowing if the message will succeed.

Now suppose A receives a message from C, containing a reference $@t$. If this message reaches A's mutator, then a reference will exist from A to C. To account for this potential reference, a stub t_A is created, in ERT_A, *before* delivering the message to the mutator:

$$t_A \rightsquigarrow C$$

If t_A already exists, a new one is not created (but its location information may be updated if more recent; see [4]). No matter how many references $@t$ exist in space A, a single stub t_A exists.

4.3 Object Migration

Let us now consider some object x which migrates from space A to B (see Figure 3) in a message:

$$A \rightarrow B : \mathsf{mutator}\{\ldots, x, \ldots\}$$

There may exist references toward x (in space A, or from any other space into A). Therefore, we consider a potential reference is created from A to x in space B. *Before* transmitting the message containing x from A, install a stub $x_A \rightsquigarrow B$.

When the message is received by B, a potential reference exists from A to B; create an entry containing $(A@x)$ in ODT_B, *before* delivering the message to the mutator.

That object x may itself contain a reference to an other object y. Then, in addition to the migration protocol above, we execute the normal procedure for transmitting $@y$ from A to B (see Section 4.2). If any indirections form, they will be eliminated by the finder.

4.4 Local Garbage Collection

Starting from a minimal ODT, the mutator's actions can only add entries into the ODT, which therefore either remains minimal or becomes a superset of its minimal value. Therefore, local garbage detection is correct, i.e. either exact or conservative.

For instance, in Figure 2, let us now run A's local garbage collector. Object z is not reachable from the local root R_A, but is reachable from ODT_A. It is not known if it is globally reachable (in fact, it is not), but conservatively it will be considered live. t_A is reachable by the dashed arrow. If the dashed arrow is removed, then t_A and t are garbage; t_A is removed by A's local GC, whereas t is removed by the protocol in the next section.

4.5 Interaction Between Local Collectors

If only local garbage detection exists, then the ODTs will grow without limit, possibly causing local GC to become inoperative. A global garbage detection protocol is necessary, to remove useless entries in ODTs.

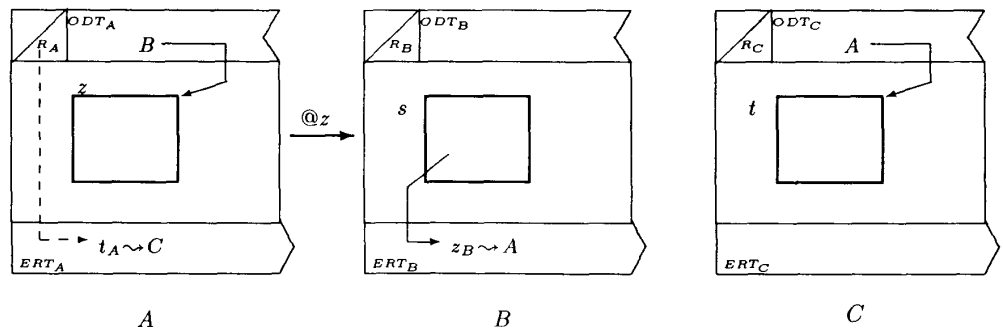

Figure 2: Sending and Receiving References

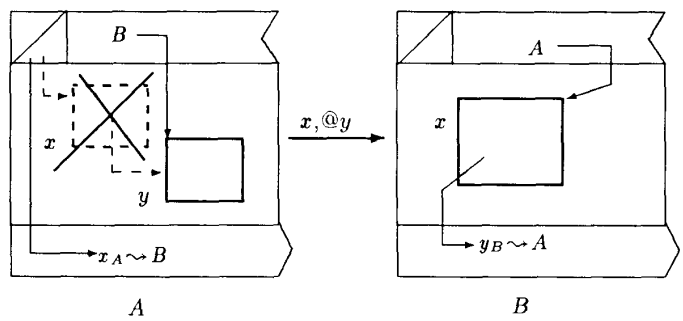

Figure 3: Migration of an Object Carrying a Reference

Suppose we remove the dashed arrow in Figure 2, from R_A to t_A. Then the next execution of A's local detector will recognize t_A to be garbage. t_A will be removed. To announce stub removal, spaces exchange background control messages such as:

$$A \rightarrow B : \mathsf{ERT}\{ERT_{A|B}\}$$

where $ERT_{A|B}$ denotes the subset of stubs of ERT_A pointing to space B. If some ODT entry $ODT_B[i] = (A@x)$ does not appear in $ERT_{A|B}$, then that entry can be removed. At the next execution of C's local GC, t will be recognized as garbage.

Some space A will send ERT messages only to those spaces which, either currently appear in ERT_A, or were recently removed from it. This is not enough, since after the last reference to some space B is removed, B might never receive $ERT_{A|B}$ and hence never collect the corresponding ODT_B entry. The solution is for B to request the ERT message from those spaces which appear in ODT_B but from which B has not heard from recently:

$$B \rightarrow A : \mathsf{request\_ERT}\{\}$$

to which the receiver A responds with a normal ERT message as above. This ensures that all garbage ODT entries are eventually deleted.

To sum up, global garbage detection occurs by pairwise cooperation between spaces. Local GC conservatively updates the ERT; each such ERT is in turn used to conservatively create new versions of the ODTs. Local GC contributes to clean up the local ERT, and hence remote ODTs.

This discussion shows that the garbage detection algorithm sketched above is indeed correct (a conservative estimate of live objects is maintained at all times) and does eventually find some garbage. However, as we will see next, it does not detect inter-space cycles of garbage.

4.6 Removing Inter-Space Cycles of Garbage

We now turn to the problem of inter-space cycles of garbage. Consider object x in space A, containing a reference to y in space B; y in turn contains a reference to x. Even if x and y are not accessible from any local root, reclaiming A and B necessitates an extension of the protocol. A simple solution [1] is to migrate the cycle to a single space, where it will be collected by the normal operation of local GC. An alternative is Hughes' algorithm [7]. Each of these alternatives would do the job, but each has its own drawbacks: some objects cannot migrate, and Hughes' algorithm is makes no progress if a single space disconnects. Therefore, in our implementations we plan to combine them both.

Let us explain the migration strategy a bit. During local GC, two marking "colors" are used. An object accessible from the local root is marked green, whereas one accessible only from the ODT is marked red. At the end of local GC, a red object may be migrated to some space which references it, by the own authority of the local GC. This has the desirable property of improving locality.

In theory, this algorithm could "ping-pong" (x is migrated to B while at the same time y is migrated to A, and vice-versa indefinitely often). In practice, because local GCs occur at different instants, this will occur very rarely; even if it ever does, the second strategy will eventually eliminate the cycle.

4.7 Loss and Duplication of Mutator Messages

If the mutator is capable of tolerating lost messages, our protocol tolerates loss also. To see this, remember that its key points are: first, maintaining a conservative estimate of each space's ODT; and second, using local GC to remove garbage entries from ERTs and hence from ODTs.

Even in the presence of message loss, the conservative nature of ODTs is maintained, since an ODT entry is made before the actual message transmisssion, and removed only after it is known to point to garbage. Therefore, garbage detection remains correct.

Let us now show that garbage detection remains effective. Local GC remains as before. We need to show, additionally, that even if an object becomes garbage by loss of a message containing the last reference to it, it will be detected. Consider the following example: object x is in space A. A single reference to x exists, from R_A. A sends

$$A \rightarrow B : \mathsf{mutator}\{@x\}$$

thus installing

$$ODT_A[12] = (B@x)$$

then deletes the reference from R_A to x. The message is lost; x is now garbage. B later sends the ERT message:

$$B \rightarrow A : \mathsf{ERT}\{ERT_{B|A}\}$$

where $ERT_{B|A}$ contains no entry for x. This causes $ODT_A[12]$ to be deleted.

Duplication of a message causes no problem. Its only effect will be to redo the same action twice. Garbage detection protocol actions are all idempotent.

4.8 Deletion and Termination

4.8.1 Deletion Protocol

The mutator may explicitly delete an object (even though it is reachable) by setting its state to **deleted**, where it contains no data and no references. This is called *normal* deletion.

As the result of unexpected mutator behaviour, or of a crash, an object may also be lost entirely. Since the effect is similar to deletion, we call this case *abnormal deletion*. We augment the finder with the following abnormal deletion protocol, to recover back to the normal case. Consider the resolution of $@x$ in space A:

- If $@x$ points to an object, resolution succeeds immediately.
- If $@x$ points to a stub $x_A \rightsquigarrow B$, and B has no corresponding ODT entry ($A@x$), then x has been deleted abnormally; the protocol then recreates it in the **deleted** state.
- If however B is disconnected (see below) then the protocol waits for it to either reconnect or terminate, and then tries again.

We deal next with the case where B terminates.

4.8.2 Disconnection and Termination of Spaces

Just as the mutator can delete an object, it may terminate a space normally. Unexpected behaviour may also cause a space to terminate *abnormally.*

The termination concept abstracts events such as a process exiting or being killed, decommissioning a machine, or reformatting a storage volume. When a space

terminates normally, all the objects and references it contains are deleted. When it terminates abnormally, and references to it continue to exist, the abnormal termination protocol of the finder, described hereafter, recovers to the normal case.

When it is not possible to communicate with a space, and it is not known whether it has terminated or not, it is said to be *disconnected.* Eventually a disconnected space either reconnects or terminates abnormally[1]. Disconnection abstracts temporary communication failures, network partitioning, site crash and recovery, or temporarily dismounting a disk volume.

Normal Termination. The normal termination protocol for space B in Figure 4 must take into account the reference from space A to t, and the indirect reference to from A to y (in space C) via B. The normal termination protocol for B is the following:

1. Remove all references from the root R_B,
2. set all local objects to the **deleted** state, but leave stubs intact,
3. perform a local garbage collection. The only objects which remain are:
   - stubs referenced from the object directory table ODT_B (indirections)
   - deleted local objects referenced from the object directory table ODT_B.
4. Cause the finder to perform indirection elimination for all remaining stubs.
5. Re-create deleted local objects in every space which refers to them.
6. Collect space B.

For performance reasons, it is preferable to await an acknowledgment of steps 4 and 5.

Disconnection. While some space B is disconnected, no references into A can be resolved (the finder must wait), and garbage detection is partially disabled, since ODT entries of the form $(B@x)$ cannot be removed.

A disconnected space will eventually either reconnect or terminate. If it reconnects, waiting protocols may proceed. If it terminates, we run the abnormal termination protocol below.

[1] It is impossible to distinguish, using messages alone, between a temporary disconnection and a permanent abnormal termination. In the latter case, some external mechanism or user intervention is necessary to force the correct outcome.

Abnormal Termination Protocol. Consider now the abrupt, abnormal termination of B in Figure 4. All of its contained objects and references are lost, as well as ODT_B and ERT_B. Stub t_A is now a dangling reference. Object z in space C is now garbage; y is not, being reachable from A. In the absence of the information lost in B's termination, C cannot distinguish between these two cases. In particular it is incorrect to assume ERT_B is empty, since this could cause y to be incorrectly collected.

A small addition to the protocols accounts for this problem. When the finder encounters a stub pointing to a space B which terminated abnormally, B is re-created in a special *zombie* state. B cannot send out its (now empty) ERT to other spaces, and terminate, until it is safe that all indirections through it have been eliminated by the finder, i.e. until receiving an empty $ERT_{D|B}$ from *every* other non-zombie space D. Therefore B sends request_ERT to all non-zombie spaces and awaits an empty ERT reply.When it has received them all, B can reply to incoming request_ERT messages with its now empty ERT. B can be collected (using the detection protocol at the next higher level of the space hierarchy hinted in Section 3) when referred from no more ERTs or ODTs.

Discussion. We assume crashes are fail-stop, therefore the only consequence of a crash is temporary disconnection, loss of volatile memory, and halt of computation. Objects and references stored only in volatile memory disappear; objects and references in non-volatile memory persist across crashes, and become active again when the space recovers and reconnects. Our problem is to ensure that ERTs and ODTs remain consistent through the crash and recovery, that garbage detection of non-crashed spaces continues as unperturbed as possible, and that no objects which will be reachable after recovery are incorrectly believed to be garbage during the crash.

During the time that space B is crashed, its set of live objects does not change. B also ceases to send messages entirely; therefore, for the duration of B's crash, object directory table entries in other spaces A, C, etc., containing something like $(B@x)$, will not be collected. ODT_C will continue to contain a superset of the objects potentially reachable from B; hence C's garbage detection remains correct. The collecting of objects potentially reachable from B is frozen; however the collecting of objects not reachable from B continues undisturbed.

Without loss of generality, we assume there are only two possible outcomes for the crash of some space B:

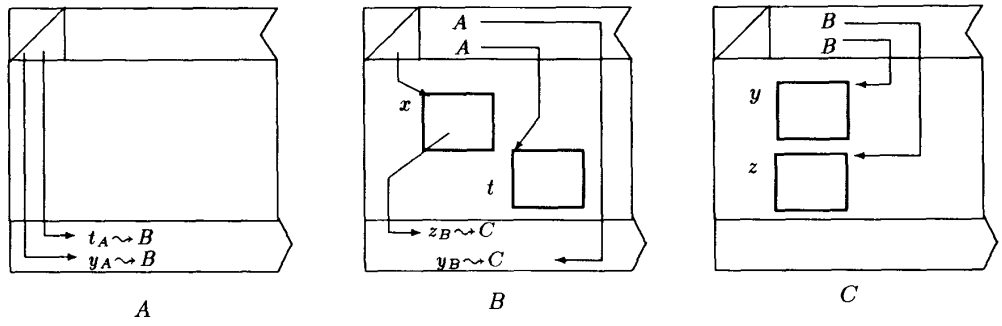

Figure 4: Termination of a Space

1. B terminates. All the objects it contained at the time of the crash are deleted. ODT_B and ERT_B are lost. B is called a *volatile space.*

2. B recovers and reconnects. An object it contains either persists or is deleted. Its object directory table ODT_B persists. B is called a *persistent space.*

Let us first consider the recovery of a persistent space. The important assumption is that, even though some of the objects it contains may be lost (i.e. deleted abnormally), its object directory table persists, hence ODT_B continues to contain a superset of its remotely reachable objects, and the garbage detection protocol remains correct. Unused ERT entries will be recovered by the next run of the local garbage collector; therefore the garbage detection protocol does continue to detect garbage.

When a volatile space crashes, all information about the objects it contained is lost. If any references to them remain elsewhere in the system, run the abnormal termination protocol to recover.

5 Correctness Considerations

Although we do not prove formally that our protocol is correct, we can give some informal evidence. It is both *safe* and *live*; i.e. it is true that only unreachable objects are collected, and that every unreachable object is eventually collected. Both these properties are based on the corresponding correctness properties of the local garbage collectors.

Let us first examine the safety property. We will show that at all times the set of uncollected objects is a superset of the set of reachable objects.

Every ODT is initially empty. Every time a new object could be remotely accessible, it is added to the ODT. An entry is removed from the ODT only when the corresponding remote garbage collector has declared it inaccessible. Therefore at all times, every ODT contains a superset of those local objects which are remotely accessible. Garbage collection starts from the local root and the ODT; therefore local uncollected objects form a superset of the set of local reachable objects (from the local or from a remote root). This implies in turn that the ERT is a superset of remote objects reachable through this space, which means that remote ODTs are indeed a superset. Since local GC starts from the union of the local root with the (conservatively estimated) ODT, all non-reachable local objects are true garbage. Since the global set of uncollected objects is simply the union of all local sets, it follows that the global set is a safe superset.

Let us now examine the liveness property. Each local GC cleans the ERT of useless stubs. In turn, ERTs are used to clean the ODTs, yielding successively better estimates. However, an ODT may possibly never become minimal, because of simultaneous mutator activity.

Garbage which is not referenced through an ODT will be eventually collected by the local garbage collector. Therefore, one only needs to consider strongly connected subgraphs of garbage passing through ODTs and ERTs. Any acyclic subgraph has a root; being garbage it is unreachable from the local root; being the root of the subgraph it is unreachable from the subgraph; therefore it is local garbage and will be collected by the garbage collector, as well as all its descendants. A cyclic subgraph will be eventually migrated to a single space, where it will be collected. Therefore the protocol is live.

6 Performance Considerations

The above protocol is still being implemented, so it is as of yet not possible to give any hard performance figures. Here we attempt a qualitative performance characterization.

A local detector never has to wait for a remote one, nor does it ever wait for an event of the global garbage detection protocol or of the object finder. Conversely, neither the object finder nor the global protocol never need to wait for a local GC.

The only synchronization requirement is that installation of new information (addition, update, or deletion of an entry), in an ODT or an ERT must be an indivisible operation. Resilience to failure and permanence are not required[2], hence atomicity (in the classic sense of atomic transactions) is not necessary.

Even while new information is being installed in an ERT or an ODT, the global and local detectors may continue to operate on an old version.

Thus, the detectors run fully parallel with each other, and with remote mutators. A detector may or may not operate in parallel with its local mutator, depending on the local GC algorithm used.

Let us now give a qualitative evaluation of the overhead with respect to a distributed system with local garbage collectors only.

We assume the existence of an object finder. Our protocol adds no algorithmic or message cost to its normal operation. Efficient operation of garbage detection assumes the finder eliminates indirections; this is desirable in any case.

Our protocol is based on the existence of an External Reference Table and an Object Directory Table per space. Normal functioning of the finder also requires these tables, or something equivalent. However some cost is added by our protocol. First, the ERT is often managed as a cache, i.e. it is allowed to be incomplete. In contrast, we require it to contain the complete list of outgoing references; but in fact this is necessary to the deletion protocol, not to the garbage detection protocol itself. In other words, this cost is brought by the requirement of fault-tolerance (support for abnormal deletion), not by garbage detection. Second, the ODT contains one entry per (possibly) referred object per space (possibly) referring it, instead of one entry per referred object. If locality is poor, this could amount to a large memory cost. Such worse-case behaviour is fought by migrating objects to enhance locality, and by a hierarchical structure of spaces.

[2]But see the definition of a persistent space in Section 4.8.2.

The message overhead is negligeable. The most important information is piggy-backed on top of existing mutator messages. The remaining background exchange of control messages can be made as infrequent as desired (at the cost of slower garbage detection).

One final cost is due to migration of objects on cycles of garbage. It has been already pointed out that such migration is a side-effect of migrating objects to their point of use, to enhance locality.

7 Conclusion

We presented a distributed garbage detection protocol. It is based on weak, realistic assumptions, making it usable for a general-purpose object-support system. Its limiting assumptions are that crashes are fail-stop, and that messages are delivered (if at all) uncorrupted, in finite time.

Until recently, garbage collection has been often judged too language-dependent, too complex and too costly for general-purpose systems. However object-support systems need the valuable service of garbage collection. Our approach is to provide a generic service for distributed garbage detection, building upon existing, language-dependent, local garbage collectors. The cooperation between local activity (mutators and collectors), and the global detection protocol, is limited to simple interactions to maintain the Object Directory and External Reference Tables.

Our protocol is based on any standard local tracing garbage collector. It is simple and deals gracefully with common error occurences of real distributed systems.

Scalability is an important property in real distributed systems. In our protocol, garbage collection is done locally, and there is no global mechanism (e.g. no global synchronization). Moreover, we rely only on local information and information exchanged between pairs of sites. For all these reasons, garbage collection is parallel, and our protocol is scalable to very large systems.

Unfortunately, the abnormal termination protocol presented here does not scale (a zombie must communicate with every other non-zombie space in the system); this problem will be addressed in a future paper.

Many of the techniques we use are well known. Our contribution has been essentially to put them together, to integrate them coherently with the rest of the system (especially the finder), at a low level of the system, independently of a particular application, language or communication protocol.

The protocol presented here has still to be tested in practice. Two implementations are under way. The first prototypes the protocol on a multiprocessor Lisp system. In addition to the protocol described above, this implementation deals with object replication. Failures are simulated for the purpose of testing the protocol design. This implementation is currently being finalized. If it confirms the qualitative performance characterization above, INRIA project SOR will undertake a full implementation on a distributed object-support operating system currently under design.

The second is for the multiprocessor object-support database system EOS, being specified at INRIA project Sabre. In this version, a space is not confined to a single machine, but can be pagewise replicated among multiple readers and writers. This necessitates only a small addition to the base protocol. Moreover, because failures are masked by the database's transaction system, the interaction between collectors degenerates to a simple reference-count protocol.

Acknowledgments

This work was done together with Olivier Gruber, of INRIA/Sabre, and David Plainfossé, of INRIA/SOR. We wish to thank our colleagues of INRIA for their helpful comments and discussions, in particular Bernard Lang, Jean-Jacques Lévy, and Damien Doligez.

References

[1] P. B. Bishop. Computer systems with a very large address space, and garbage collection. Technical Report MIT/LCS/TR-178, Mass. Institute of Technology, MIT Laboratory for Computer Science, Cambridge MA (USA), May 1977.

[2] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 147–158, Litchfield Park, Arizona USA, December 1989. ACM.

[3] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[4] Robert Joseph Fowler. Decentralized object finding using forwarding addresses. Technical Report 85-12-1, Department of Computer Science, University of Washington, Seattle, WA (USA), December 1985.

[5] Olivier Gruber and Patrick Valduriez. Uniform object management for parallel database servers. In *Data Management and Parallel Processing*. Chapman and Hall, London, 1990.

[6] Sabine Habert, Laurence Mosseri, and Vadim Abrossimov. COOL: Kernel support for object-oriented environments. In *ECOOP/OOPSLA'90 Conference*, volume 25 of *SIGPLAN Notices*, pages 269–277, Ottawa (Canada), October 1990. ACM.

[7] John Hughes. A distributed garbage collection algorithm. In Jean-Pierre Jouannaud, editor, *Functional Languages and Computer Architectures*, number 201 in Lecture Notes in Computer Science, pages 256–272, Nancy (France), September 1985.

[8] Won Kim et al. Features of the Orion object-oriented database system. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, 1989.

[9] Rivka Ladin. *A Method for Constructing Highly Available Services and a Technique for Distributed Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (USA), May 1989.

[10] Barbara Liskov and Rivka Ladin. Highly-available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the 5th Symposium on the Principles of Distributed Computing*, pages 29–39, Vancouver (Canada), August 1986. ACM.

[11] Marc Shapiro, Yvon Gourhant, Sabine Habert, Laurence Mosseri, Michel Ruffin, and Céline Valot. SOS: An object-oriented operating system — assessment and perspectives. *Computing Systems*, 2(4):287–338, December 1989.

[12] Marc Shapiro, Olivier Gruber, and David Plainfossé. A garbage detection protocol for a realistic distributed object-support system. Rapport de Recherche 1320, Institut National de la Recherche en Informatique et Automatique, Rocquencourt (France), November 1990.

[13] Fernando Velez, Guy Bernard, and Vineeta Darnis. The O_2 object manager: an overview. Technical Report 27-89, GIP-Altaïr, Rocquencourt (France), February 1989.

[14] Stephen C. Vestal. *Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming*. PhD thesis, Dept. of Comp. Sc., U. of Washington, Seattle WA (USA), January 1987. U. of Washington Tech. Report 87-01-03.